

# A Simple Hybrid GRASP-Evolutionary Algorithm for CSPs

Manuel Cebrián and Iván Dotú

Departamento de Ingeniería Informática  
Escuela Politécnica Superior  
Universidad Autónoma de Madrid

**Abstract.** There are several evolutionary approaches for solving random binary CSPs. In most of these strategies we find a complex use of information regarding the problem at hand. Here we present a hybrid GRASP-Evolutionary Algorithm that outperforms previous approaches in terms of effectiveness and compares well in terms of efficiency. Our algorithm is conceptual and simple, featuring a GRASP-like mechanism for genotype-to-phenotype mapping; moreover, it faces the problem as a search for an optimal variable ordering, a novel approach within this framework. Therefore, we provide a nice algorithm that harnesses generality while boosting performance.

## 1 Introduction

Random binary CSPs is a widely used benchmark within the constraint programming community as an efficiency test for several algorithms and solvers. However, we can also find a wide spectrum of evolutionary approaches for solving random binary CSPs. In [1] we find a comprehensive comparison of those methods, such as SAW [2], Glassbox [3], MID, CCS, . . .

This paper presents the hybrid evolutionary algorithm GA-GRASP<sub>Vo</sub> for solving random binary CSPs. The algorithm is conceptual, simple and uses a key modeling based on the ideas in [4]. GA-GRASP<sub>Vo</sub> specifically applies the idea of a GRASP-like mechanism to perform genotype-to-phenotype mapping for solving random binary CSPs. Moreover, a novel approach to face the problem is introduced: our population specifies a variable ordering, instead of a variable assignment, which is the most common representation in the literature.

We also provide a comparison with two of the most successful state-of-the-art evolutionary algorithms as shown in [1]. Our simple algorithm outperforms the best approach in terms of effectiveness while outdoing it also in terms of efficiency. Moreover, it compares with the best approach in terms of efficiency, while outperforms it in terms of effectiveness.

The main contributions of this research are:

- A novel representation which focuses on finding an optimal variable ordering, and that makes use of the ideas in [4] for a GRASP-like genotype-to-phenotype mapping.

- A general evolutionary algorithm which can be easily suited to solve any kind of CSP problems without considerable implementation effort.
- Surprising results that outperform and compare with the best evolutionary algorithms which usually involve complex heuristics and fitness adjustment functions.
- Showing that a simple algorithm can yield outstanding results if an appropriate modeling is chosen; therefore, stating the importance of representation in evolutionary strategies.

The rest of the paper is organized as follows: first we briefly introduce constraint satisfaction problems for Evolutionary Algorithms. We then present the Hybrid Algorithm and its GRASP-like representation. The following sections are devoted to the experimental comparison with other methods: in section 4 we define our test suite problem (random binary CSPs), in section 5 we introduce the methods to be compared against our algorithm, section 6 describe the measures we use for the comparison, and section 7 shows the experimental results and the comparison itself. The paper ends with some conclusions and future work.

## 2 Constraint Satisfaction Problems and EAs

In a *constraint satisfaction problem* (CSP) we are given a set of variables, where each variable has a domain of values, and a set of constraints acting between variables. The problem consists of finding an assignment of values to variables in such a way that the restrictions imposed by the constraints are satisfied.

We can also define a CSP as a triplet  $\langle X, D, C \rangle$ , where  $X = \{x_1, \dots, x_n\}$  is the set of variables,  $D = \{D_1, \dots, D_n\}$  is the set of nonempty domains for each variable  $x_i$ , and  $C = \{C_1, \dots, C_n\}$  is the set of constraints. Each constraint is defined over some subset of the original set of variables  $\{x_1, \dots, x_n\}$  and specifies the allowed combinations of these variable values. Thus, solving the CSP is equivalent to finding a complete assignment for the variables in  $X$  with values from their respective domain set  $D$ , such that no constraint  $C_i \in C$  is violated.

The evolutionary framework presents the issue of constraint handling: constraints can either be handled directly or indirectly [5].

- **Indirect handling** involves transforming the constraint into an optimization objective which the EA will pursue; while,
- **Direct handling** leaves the constraint as it is, and enforces it somehow during the execution of the algorithm.

Direct handling is not oriented for EA due to the lack of an optimization function in the CSP, which will result in no guidance towards the objective. Thus, indirect handling is the best suited approach for EA, although a mixed strategy where some constraints are enforced and some are transformed into an optimization criteria is suited as well.

### 3 The Hybrid Evolutionary Algorithm

We now turn to the hybrid evolutionary (HE) algorithm for solving CSP problems. The algorithm maintains a population of GRASP parameters and performs a number of iterations until a solution is found. Each iteration selects two individuals in the population and, with some probabilities, crosses and/or mutates them. The next population will be obtained in an elitist fashion.

Following the framework presented in [1], our algorithm consists of a generational evolutionary model with an elitist selection of the new generations, the fitness function is based in a variable penalty strategy with weights  $w_i = 1$ , the recombination operator is that of a one-point crossover, the mutation operator selects for each  $s_i$  (each single GRASP parameter in the vector), with a chance of 0.6 uniformly random a new value, the parent selection is performed in a binary tournament fashion and the constraint handling is purely indirect. It presents neither fitness adjustments or use of heuristics; and the representation is indirect using GRASP parameters. Each of these characteristics is now reviewed in more detail.

*evolutionary model* the algorithm consists of a generational strategy where new populations are selected in an elitist fashion, which means that in each new generation, the population is calculated by maintaining the best individuals among the previous population and the offspring.

*Fitness function* in order to calculate the fitness of a certain individual, we will take into account how many variables are in conflict with the rest. Thus, the fitness function would be as follows:

$$f(s) = \sum_{j=1}^n v(s, C^j), \text{ where,}$$
$$v(s, C^j) = \begin{cases} 1, & \text{if } s \text{ violates at least one } c \in C^j; \\ 0, & \text{otherwise.} \end{cases}$$

*Crossover* the HE algorithm uses a one-point crossover for crossing two individuals  $\sigma_1$  and  $\sigma_2$ . It selects a random number  $k$  in  $1..n$ . The child is obtained by selecting the first  $k$  genes from  $\sigma_1$  and the remaining  $n - k$  genes from  $\sigma_2$ .

*Mutation* the mutation here is achieved by randomly selecting a new value for each single GRASP parameter in the vector that defines the individual, with a 0.6 probability.

*Parent selection* each iteration selects two individuals in the population (the parents). This is performed in a binary tournament fashion: randomly select two individuals from the population and choose the best of them; do the same for the second parent.

```

procedure GRASP(maxIt,seed)
1. Read_Input()
2. for k=1,..., maxIt do
3.   Solution  $\leftarrow$  Greedy_Randomized_Construction(seed);
4.   Solution  $\leftarrow$  Local_Search(Solution);
5.   Update_Solution(Solution);
6. end;
7. return Best_Solution;
end GRASP

```

**Fig. 1.** The GRASP pseudocode

```

procedure Greedy_Randomized_Construction(seed)
1. Solution  $\leftarrow$   $\emptyset$ 
2. Evaluate the incremental costs of candidate elements
3. While Solution is not complete do
4.   Build the restricted candidate list RCL
5.   Select element s from RCL at random
6.   Solution  $\leftarrow$  Solution  $\cup$  {s};
7.   Reevaluate the incremental costs;
8. end;
9. return Solution;
end Greedy_Randomized_Construction

```

**Fig. 2.** The Greedy Randomized Construction pseudocode

## Representation

This is the most important feature in our algorithm, in fact, the rest of the characteristics are common in simple evolutionary schemes. However, the representation of the CSP is a key factor in the algorithm efficiency. In order to define our implementation, we must introduce some basic concepts.

*GRASP* The GRASP (Greedy Randomized Adaptive Search Procedure) meta-heuristic can be viewed as an iterative process, each iteration consisting of two phases: construction and local search ([6]). The construction phase builds a solution whose neighborhood is investigated by the local search procedure. During the whole process, the best solution is updated and returned at the end of a certain number of iterations. Figure 1 illustrates the basic GRASP procedure.

Though GRASP techniques are deeply studied, and several extra features have been developed, we are most interested in the construction procedure, which is depicted in figure 2. The features of the constructed solutions are going to be the key element in our representation.

*GRASP parameters* As you can see in figure 2, a key element in the construction procedure is the selection of the elements. Due to space constraints it is not possible to review all the existing techniques related to this step, but we will refer the reader to [6] for a comprehensive review, and also to [4] as we will use the same approach for our representation. In [4] a Hybrid GRASP - Evolutionary algorithm for finding Golomb rulers is introduced. Our representation makes the same use of the GRASP features as in the mentioned algorithm.

Thus, our population is going to be a set of individuals which are vectors of *GRASP parameters* in a similar manner as in [4]. The value of each parameter reflects the decision to take in this step, forcing us to choose the decision ranked in the position indicated by the value. Decisions are ranked according to some quality criteria, thus, a parameter value 0 will involve taking the “best” decision. A vector with all parameters set to 0 corresponds to a plain greedy strategy.

*Parameters concordance* What we have not explained yet is how these parameters correspond to a variable assignment for a CSP.

It would be easy to think that these parameters correspond somehow to the values we will assign to the variables, being these variables ordered with some static heuristic. Looking at this possibility in detail, we realize that it would make no sense, since we will be choosing values which violate constraints unless the parameter is 0; or more than one of the values yield no violations, but in this case we would not have a clear way to rank them. This seems to be a problematic obstacle for this option.

Reviewing known facts within the constraint programming community, and as it is shown in [7], the ordering heuristic for assigning variables is a key factor in quickly finding a feasible solution. Based on that, we will assume that it is possible to assign the variables in a certain order such that we will be able to find a solution assigning values that do not generate conflicts.

What follows is that we are going to transform the problem of finding values for the variables into finding an optimal ordering for the variables that will yield a feasible solution. Thus, our vector of GRASP parameters will allow us to choose, among the ranked variables, which one we want to instantiate next. The variables will be dynamically ranked using the dom/degree ordering heuristic [7] (quality criteria), which gives more weight to variables with few available values in its domain, and that take place in a greater amount of constraints. The values that the parameters can take, will fall within the range  $[0, n - pos_i]$ , where  $pos_i \in 1..n$  is the position of the given parameter within the vector. In this case, the last parameter will always be 0, since there is just one variable left to assign.

It is worth mentioning that, opposite to [4], we allow non feasible instantiations. This follows immediately from the fact that we are considering a feasibility problem, instead of an optimization problem. In the latter we are searching for the best feasible solution, hence, we can restrict the search to feasible solutions; while in the former we are searching for the best unfeasible solution, which corresponds to a feasible solution (the one with less constraint violations).

```

1. GA-GRASPVo(csp)
2.   forall  $i \in 1..populationSize$ 
3.      $\Sigma \leftarrow \Sigma \cup \{\text{RANDOMCONFIGURATION}(csp.n)\};$ 
4.    $g \leftarrow 0;$ 
5.   while  $g \leq maxGen$  &  $v(\Sigma) > 0$  do
6.      $i \leftarrow 0;$ 
7.      $\Sigma^+ \leftarrow \emptyset;$ 
8.     while  $i \leq populationSize$  do
9.       select  $(\sigma_1, \sigma_2) \in \Sigma;$ 
10.      with probability  $P_c$ 
11.         $\sigma^* \leftarrow \text{crossover}(\sigma_1, \sigma_2);$ 
12.        if  $v(\sigma^*) == 0$ 
13.          return  $\sigma^*;$ 
14.         $\sigma^* \leftarrow \text{mutate}(\sigma_2, P_m);$ 
15.        if  $v(\sigma^*) == 0$ 
16.          return  $\sigma^*;$ 
17.         $\Sigma^+ \leftarrow \Sigma^+ \cup \{\sigma^*\};$ 
18.         $i \leftarrow i + +;$ 
19.       $\Sigma \leftarrow \text{select}(\Sigma^+, \Sigma);$ 
20.       $g \leftarrow g + 1;$ 

```

**Fig. 3.** Algorithm GA-GRASP<sub>V<sub>o</sub></sub> for CSP problems

### 3.1 The Hybrid Algorithm

We are now ready to present the HE algorithm GA-GRASP<sub>V<sub>o</sub></sub> which is depicted in Figure 3. Lines 2-4 perform the initializations. In particular, the population is randomly generated in lines 2-3 and the generation counter  $g$  is initialized in line 4.

The core of the algorithm is in lines 5-20. They generate new generations of individuals for a number of iterations or until a solution is found. The new generation is initialized in line 7, while lines 8-16 create the new generation. The new individuals are generated by selecting the parents in line 9, applying a crossover with probability  $P_c$  (lines 10-11), and applying a mutation with probability  $P_m$  (line 14). The new individuals are added to the new population in line 17. The current population is selected among the previous and the new population in line 19. Note that after crossover and mutation we need to calculate the cost of the individual in order to detect solutions and/or keep track of the cost in order to properly select parents and next population.

## 4 Our benchmark: random binary CSPs

In this papers we consider random binary constraint satisfaction problems, since their properties in terms of difficulty to be solved have been well-understood and hence such problems have been used for testing the performance of algorithms

p	E(solutions)
0.24	1707299.07
0.25	258652.614
0.26	38984.6092
0.27	5600.99655
0.28	838.870129
0.29	125.400589
0.30	19.6420135
0.31	2.79148238
0.32	0.42173145
0.33	0.06618763

**Fig. 4.** The Smith’s conjecture prediction of the number of solutions as a function of  $p$ .

for solving binary CSPs. In [8] it was shown that any CSP can be equivalently transformed to a binary CSP, thus no generality is lost.

Various problem instance generators have been developed for the class of binary CSPs, based on several theoretical models. All of these models are parametrized by  $n$ ,  $m$ ,  $D$ , and  $k$ , where  $n$  is the number of variables,  $m$  is the number of constraints,  $D$  is the number of values in each domain and  $k$  is the arity of each constraint. In a binary constraint network, the value of  $k$  is fixed to 2.

There are four traditional models, called A, B, C and D developed from a general framework presented in [9] and [10], but they are recently being replaced by a newer model proposed by Achioptlas *et al.* [11] which does not suffer from the deficiencies underlying the other models<sup>1</sup>. This model is usually specified as  $E(n, p, D, k)$  with  $p$  defined as  $p = m \binom{n}{k} D^k - 1$ . The model  $E$  works by choosing uniformly, independently and with repetitions conflicts between two values of two different variables.

Our test suit consists of 250 solvable problem instances available on the Web [12] and used also as a benchmark in [1]; they are generated using the model  $E(20, p, 20, 2)$  with 25 solvable instances for each value of  $p$  in  $\{0.24, 0.25, 0.26, 0.27, 0.28, 0.29, 0.30, 0.31, 0.32, 0.33\}$ . By using the conjecture of Smith [10] we show that the range for  $p$  in model E actually runs through the mushy region. The term mushy region is used to indicate the region where the probability that a problem is soluble changes from almost zero to almost one. Within the mushy region, problems are in general difficult to solve or to prove unsolvable. In Figure 4 it can be seen that the predicted number of solutions drops below one when moving from  $p = 0.31$  to  $p = 0.32$ , just what defines the mushy region.

## 5 Related Work

There are several evolutionary algorithms focused on solving random binary CSPs [1]. Most of them use knowledge of the problem, either to develop heuristics

<sup>1</sup> The instances generated by the A to D models are unsolvable with high probability.

or to implement a fitness adjustment technique. The nice feature of our algorithm is that no knowledge about the problem is taken advantage of, hence, harnessing generality without a loss in efficiency or effectiveness.

In this section we are going to briefly introduce the two most successful approaches according to [1], which will be used later to compare against our algorithm.

*SAW* the basic idea behind the SAW (Stepwise Adaptation of Weights) algorithm lies in the way that the fitness function is evaluated. Each  $k$  evaluations<sup>2</sup> the variables causing the constraint violations in the best individual of the current population are given a high weight (penalty), because they are considered to be harder than the others. These weighted-up variables will have a greater impact in the fitness of the following evaluations. A comprehensive study of different parameters and genetic operators of SAW can be found in [2].

*Glassbox* Glass-Box works by decomposing complex constraints in two steps: elimination of functional constraints and decomposition of the CSP into primitive constraints, usually of the form  $\alpha \cdot p_i - \beta \cdot p_j \neq \gamma$  where  $p_i$  and  $p_j$  are the values of variables  $v_i$  and  $v_j$ . A common repair rule used is the following

$$\text{if } \alpha \cdot p_i - \beta \cdot p_j = \gamma \text{ then change } v_i \text{ or } v_j \quad (1)$$

Repairing a violated constraint can result in the production of new violated constraints, thus at the end of repairing process the chromosome will not in general be a solution. An extensive work on this constraint processing technique is presented in [3] and [13].

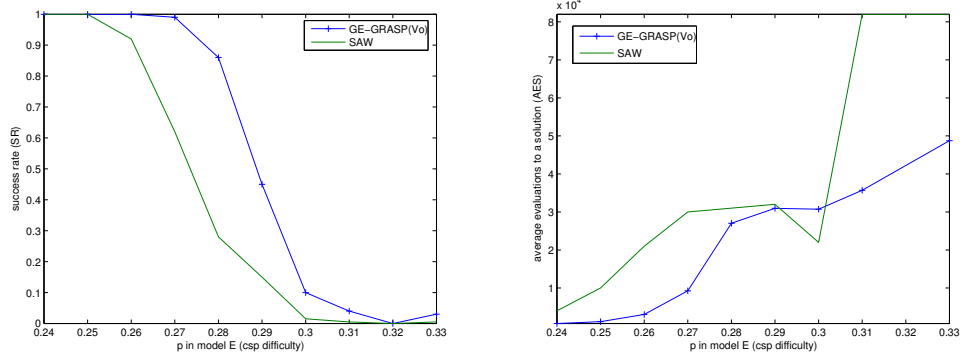
## 6 Measures of effectiveness and efficiency

Genetic algorithms are random algorithms, therefore the behavior of the optimization in a problem instance varies from execution to execution. In order to obtain a more accurate idea of the performance of an algorithm in a concrete binary CSP instance, we are going to run it 5 times for each problem instance, thus having 125 executions for each  $p$  value (5 executions for each 25 problem instances belonging to a concrete  $p$  value). The set of executions for each  $p$  value is denoted by  $S_p$ .

It remains to be defined when an execution has finished. Obviously, if the genetic algorithm finds the solution we consider the execution to be finished, however we need to set a time limit in order to avoid waiting indefinitely. Our choice of the time unit is the evaluation of an individual, i.e. each calculation of the fitness (in this case the number of constraint violations) of an individual. We define  $\theta$  as the maximum number of evaluations for each execution. Now, we can define some effectiveness and efficiency measures as a function of  $p$ .

---

<sup>2</sup> In [1] the period  $k$  is set to 25 evaluations.



**Fig. 5.** *SR* and *AES* measures from the GA-GRASP<sub>Vo</sub> and SAW algorithms.

### Effectiveness

Effectiveness is measured by the success rate (*SR*), the mean error at termination (*ME*) and the average champion error (*ACE*).

We define  $S_p^+$  as the executions of  $S_p$  that found a solution before  $\theta$  evaluations, and  $S_p^- = S_p - S_p^+$ . The *SR* over  $p$  is the percentage of runs that find a solution in no more than  $\theta$  evaluations.

$$SR(p) = 100 \frac{|S_p^+|}{|S_p|} \quad (2)$$

The error at termination (*ET*) is defined for a single run as the number of constraints violated by the best candidate solution in the population when the execution reaches  $\theta$  evaluations. If an execution finishes before  $\theta$  evaluations, then its *ET* is considered 0, thus the mean error at termination (*ME*) is defined as

$$ME(p) = \frac{1}{|S_p^-|} \sum_{s \in S_p^-} EAT(s) \quad (3)$$

We use another effectiveness measure that focuses on the convergence speed of the algorithm. We define the champion error (*CE*) as the number of constraints violated by the best individual found up to a given time (measured in evaluations) during a run, thus the average champion error is defined as

$$ACE(p, t) = \frac{1}{|S_p|} \sum_{s \in S_p} CE(s, t) \quad (4)$$

If  $s$  has finished before  $t$  evaluations, then  $CE(s, t) = 0$ .

### Efficiency

In our experiments, we use the average number of evaluations to find a solution (*AES*) in order to measure efficiency. The *AES* is the average number of

evaluations to find a solution ( $ES$ ) over the successful runs  $S_p^+$ .

$$AES(p) = \frac{1}{|S_p^+|} \sum_{s \in S_p^+} ES(s) \quad (5)$$

It is important to note that if  $S_p^+ \ll S_p^-$  then  $AES$  is statistically unreliable.

## 7 Experimental results

We have chosen the  $SR$ ,  $ME$ ,  $ACE$  and  $AES$  measures in order to compare the measures obtained from our algorithm GA-GRASP $_{Vo}$  with the same measures of the most effective and efficient algorithms analyzed in [1]. The test suite is the same: the 250 instances generated from model E which are available in [12]. The limit number of evaluations parameter is also the same, and set to  $\theta = 100000$ .

### 7.1 Effectiveness

The most important measure in evaluating effectiveness is the success rate, because the main goal of an algorithm for solving CSPs is to obtain a solution. In [1] it is shown that the overall winner regarding success rates is the SAW algorithm.

In Figure 5 we give a comparison of the  $SR$  measures between the GA-GRASP $_{Vo}$  and SAW algorithms. SAW is outperformed by GA-GRASP $_{Vo}$  in all  $p$  values. Moreover, if we consider the global success rate for all  $p$ 's, we obtain an overall  $SR$  of 55% for GA-GRASP $_{Vo}$  and 44% for SAW, which implies more than a 10% of successful executions.

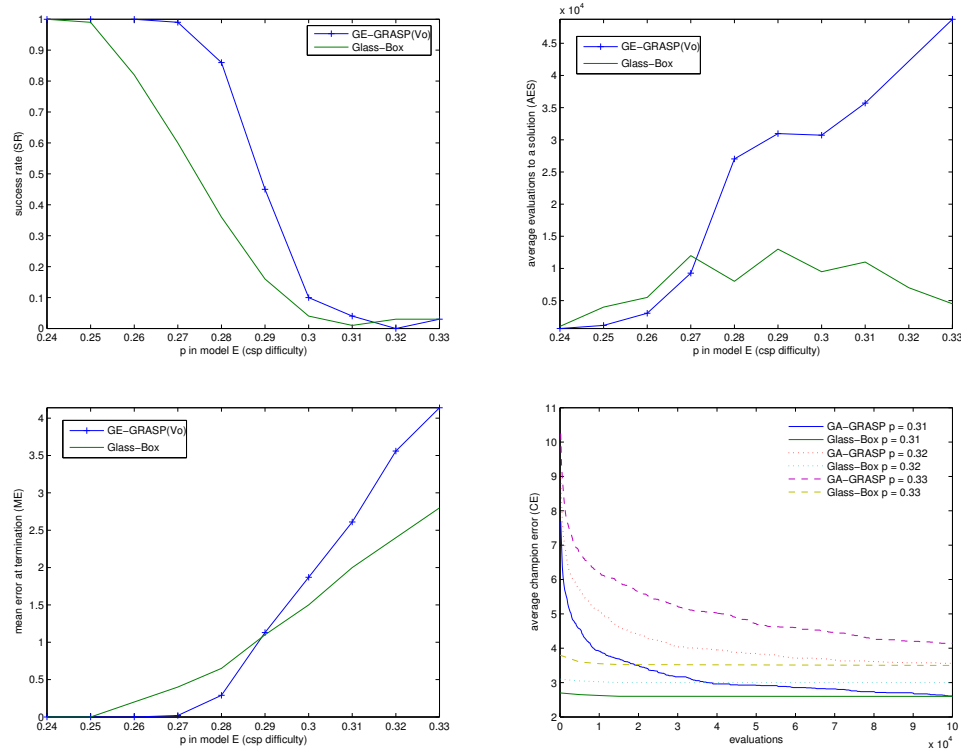
Unfortunately, due to the way in which the fitness function is computed in SAW, its  $ME$  and  $ACE$  measures cannot be compared to the same ones from GA-GRASP $_{Vo}$ . This is explained because the fitness of the SAW algorithms is not the number constraint violations, but a weights-scaled function of it.

We include an efficiency comparison in Figure 5, the average number of evaluations to a solution. In all but one  $p$  the GA-GRASP $_{Vo}$  requires less evaluations than SAW. Curiously, from  $p = 0.24$  to  $p = 0.30$  the two algorithms seem to converge, but from that points onwards the  $AES$  for SAW grows exponentially while having only a slight increase for GA-GRASP $_{Vo}$ .

### 7.2 Efficiency

In [1] it is shown that the overall winner regarding efficiency measured by the number of fitness evaluations is the Glass-Box genetic algorithm.

In Figure 6 we give a comparison of the efficiency measure ( $AES$ ) between the GA-GRASP $_{Vo}$  and Glass-Box algorithm. In the easy region (0.24 to 0.27) GA-GRASP $_{Vo}$  needs less evaluations, but is surpassed in terms of efficiency by Glass-Box in the mushy region. The average number of solutions over all  $p$ 's is



**Fig. 6.** Efficacy and efficiency measures from the GA-GRASP $_{Vo}$  and Glass-Box algorithms.

20812 for GA-GRASP $_{Vo}$  and 7889 for Glass-Box, being the last a 37% more efficient.

In efficacy terms, the two algorithms are more balanced. GA-GRASP $_{Vo}$  outdoes Glass-Box in the easy region (0.24 to 0.30), with an equilibrium in the mushy region. In overall terms the Glass-Box successfully finishes a 40% of the executions, a 15% less than GA-GRASP $_{Vo}$ .

Observing the *ME* and *ACE* of Figure 6 it can be seen that the quality of the partial solutions during the execution is slightly better for Glass-Box, specially in the mushy region, where it has 1 less violated constraint on average at the end of the run.

## 8 Conclusion and Future Work

In this paper we have presented a hybrid evolutionary algorithm for solving random binary CSPs, which yields surprising results, as it outperforms the best previous approach in terms of effectiveness, and compares with the best strategy in terms of efficiency.

Our hybrid algorithm incorporates features of GRASP, in a similar fashion as in [4], where a GRASP-like mechanism is applied to genotype-to-phenotype mapping. The rest of the algorithm is conceptual and simple, and makes no use of information regarding the problem, which harnesses generality. It also demonstrates that modeling (or representation) is a key factor in evolutionary strategies.

Moreover, we believe there is a large space for improvement. Note that no restarting technique is implemented, which usually leads to better results in previous EAs. Some preliminary results indicate that the use of learning techniques such as Lamarckian learning can boost performance significantly. Finally, we are interested in using real life CSP benchmarks in order to compare results with constraint programming techniques, and other evolutionary approaches available.

## Acknowledgments

We would like to thank Carlos Cotta and Antonio Fernández for some useful discussions on their publication [4].

## References

1. B.G.W. Craenen, A.E. Eiben, J.I. van Hemert; *Comparing Evolutionary Algorithms on Binary Constraint Satisfaction Problems* in IEEE Transactions on Evolutionary Computation, Vol. 7, Number 5; IEEE Neural Networks Society Press; 424-445; October 2003.
2. B.G.W. Craenen, A.E. Eiben; *Stepwise Adaption of Weights with Refinement and Decay on Constraint Satisfaction Problems* in Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2001; L. Spector, E. Goodman, A. Wu, W.B. Langdon, H.-M. Voigt, M. Gen, S. Sen, M. Dorigo, S. Pezeshk, M. Garzon, and E. Burke ed.; Morgan Kaufmann Publishing, San Francisco, CA; pages 291-298; July 2001.
3. E. Marchiori. *Combining constraint processing and genetic algorithms for constraint satisfaction problems* in Proceedings of the 7th International Conference on Genetic Algorithms, San Francisco, CA, 1997. Morgan Kaufmann Publishers, Inc. pages 30337.
4. C. Cotta and A. Fernández, *A Hybrid GRASP-Evolutionary Algorithm Approach to Golomb Ruler Search* in Parallel Problem Solving From Nature VIII, vol. 3242, pp. 481490, Berlin, 2004.
5. A. E. Eiben, *Evolutionary algorithms and constraint satisfaction: Definitions, survey, methodology, and research directions* in Theoretical Aspects of Evolutionary Computation, pp. 1358, 2001.
6. , *Greedy randomized adaptive search procedures* in Handbook of Metaheuristics, pp. 219249, 2003.
7. I. Gent, E. MacIntyre, P. Prosser, B. Smith, T. Walsh, *An empirical study of dynamic variable ordering heuristics for the constraint satisfaction problem* in: E.C. Freuder (Ed.), Proc. 2nd International Conference on Principles and Practice of Constraint Programming (CP-96), Springer, Berlin, 1996, pp. 179193.

8. F. Rossi, C. Petrie, and V. Dhar, *On the equivalence of constraint satisfaction problems* in Proc. 9th European Conf. Artificial Intelligence ECAI'90, L. C. Aiello, Ed., 1990, pp. 550-556.
9. P. Prosser, *An empirical study of phase transitions in binary constraint satisfaction problems* in J. Artif. Intell., vol. 81, pp. 81109, 1996.
10. B. Smith and M. Dyer, *Locating the phase transition in binary constraint satisfaction problems* in J. Artif. Intell., vol. 81, no. 12, pp. 155181, 1996.
11. D. Achlioptas, L. Kirousis, E. Kranakis, D. Krizanc, M. Molloy, and Y. Stamatiou, *Random constraint satisfaction: A more accurate picture* in Constraints, vol. 4, no. 6, pp. 329344, 2001.
12. CSP Problem Instances Using Model E (2002). Available at [http://www.cs.vu.nl/~bcraenen/resources/csps\\_modelE\\_v20\\_d20.tar.gz](http://www.cs.vu.nl/~bcraenen/resources/csps_modelE_v20_d20.tar.gz)
13. P. van Hentenryck, V. Saraswat, and Y. Deville. *Constraint processing in cc(FD)* in Constraint Programming: Basics and Trends. Springer-Verlag, Berlin, 1995.